CS436 - Cloud Computing Term Project

Ovatify on the Cloud

Github Link

Group Members:

Akif Işıtan 29354

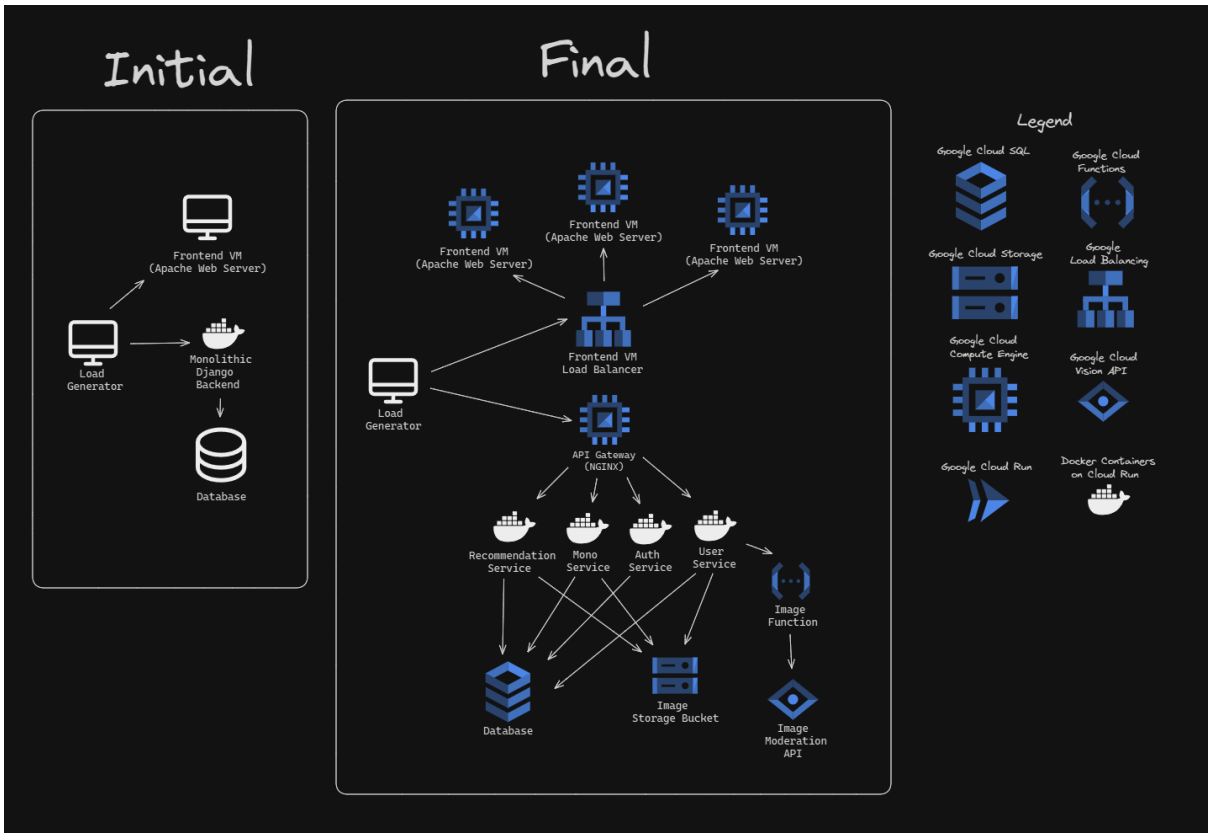
Bilal Berkam Dertli 29267

Mehmet Enes Onuş 29353

Abdulrahman Yunis 30526

Date: 26.05.2024

System Architecture



Initial Architecture

The initial system architecture consisted of a single Frontend Virtual Machine (VM) hosting an Apache Web Server, which interacted directly with a monolithic Django backend. This setup was simple, with a single database handling all data storage needs. A load generator was used to simulate user traffic.

Limitations

- **Scalability:** The monolithic architecture limited the system's ability to scale efficiently under high load conditions.
- **Flexibility:** Updates to one part of the system often required redeploying the entire backend, increasing downtime and the potential for bugs.

Final Architecture

The final architecture adopted a hybrid microservices approach, greatly improving scalability and maintainability:

- **Frontend:** Multiple Frontend VMs were used, each hosting an Apache Web Server, with a Layer 4 Load Balancer to distribute incoming traffic evenly.
- **Backend:** Decomposed into several independent microservices (Recommendation Service, Mono Service, Auth Service, User Service) each running in separate Docker containers on Google Cloud Run. This setup allows each service to scale independently based on demand.
- **API Gateway:** An Nginx API Gateway was implemented to manage and route API calls to the appropriate backend services.
- **Data Management:** A central database was retained for overarching data needs, while specific functionalities like image handling were supported by an Image Storage Bucket and an Image Function tied to a Google Cloud Vision API for content moderation.

Benefits of this setup were:

- **Scalability:** The use of microservices allows each component of the backend to scale based on its specific load, preventing overuse of resources.
- **Maintainability:** Updates or fixes can be rolled out to individual services without affecting the entire system.
- **Performance:** Load balancing across multiple frontend servers improves response times and system resilience.

Technologies Used

- **Frontend:** Svelte framework, known for its reactivity and compact size, was chosen to build a fast and responsive user interface.
- **Backend:** Django was used for its robustness and ease of integrating with other services and databases.
- **Cloud Services:**
    - **Google Cloud SQL** for reliable, scalable database services.
    - **Google Cloud Functions** for executing backend code in response to events without managing server infrastructure.
    - **Google Cloud Run** for running microservices as containers and handling auto-scaling for high loads.
    - **Google Cloud Storage** for storing images and other static assets.

    - **Google Cloud Vision API** for implementing image moderation.
    - **Google Cloud Load Balancing** to distribute user requests efficiently across server resources.
- **API Gateway:** Nginx, for its performance and reliability in handling concurrent connections and routing them to appropriate services.
- **Containerization:** Docker on Google Cloud Run for container management, ensuring each microservice is isolated, scalable, and easy to deploy.

Implementation

Frontend

Implemented using Svelte, the frontend provides an interactive user interface where users can manage playlists, add friends, and receive song recommendations. The static assets are served through an Apache Server, which lets the client browser run the downloaded Javascript and create the necessary UI. After the Javascript is executed in the client browser, it fetches data from the backend through RESTful APIs routed through the NGINX API Gateway.

Backend

The backend is structured into several microservices. Backend has a hybrid approach to microservices with 3 purpose based containers and a mono service that handles general tasks in a single container.

- **Recommendation Service:** Manages the logic for generating music recommendations based on users' and their friends' listening habits.
- **Auth Service:** Handles user authentication and session management.
- **User Service:** Manages user profiles, user preferences and interactions among users.
- **Mono Service:** General operations such as returning favorite songs, getting playlists, removing a song etc.

Each service is containerized using Docker and deployed on Google Cloud Run for easy scaling.

- **Database:** A central Google Cloud SQL database is used for data integrity and transaction management across different services.

- **Image Cloud Function:** Processes image data using Google Cloud Vision API for moderating content uploaded by users. Creating a safer environment for everyone in terms of images.

Performance Analysis

Backend

Backend service instances at Google Cloud Run perfectly scale up and down according to request counts. At our test we set 1000 peak users with 15 ramp up value which increases the load by 15 concurrent requests per second. We observed a higher average response time when increasing the load at the services. This increase was most probably due to high overhead of load balancing and high CPU Utilization at peak times.

Figure 1: Locust Test Graph of Backend

Figure 2: Authentication Service Observability Graphs

Figure 3: Mono Service Observability Graphs

Figure 4: Recommendation Service Observability Graphs

Figure 5: User Service Observability Graphs

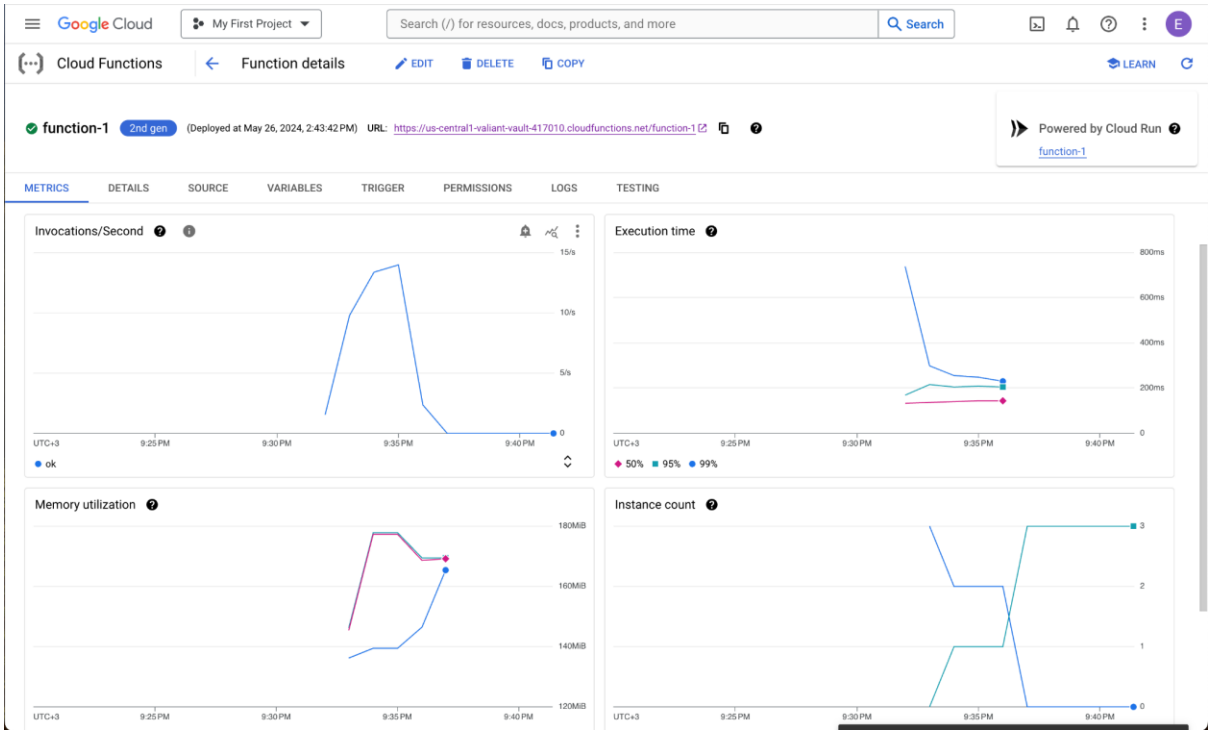


Figure 6: Cloud Function Observability Graphs

Frontend

Front-end services consisted of 3 virtual machine instances. We tested this system with a maximum of 1000 concurrent users with an increment of 15. Our choice of 3 VMs was successful in terms of handling the load and had very good latency. After a certain point, our latency increases slightly as we become memory bound instead of CPU bound.

Figure 7: Locust Test Graph of Frontend

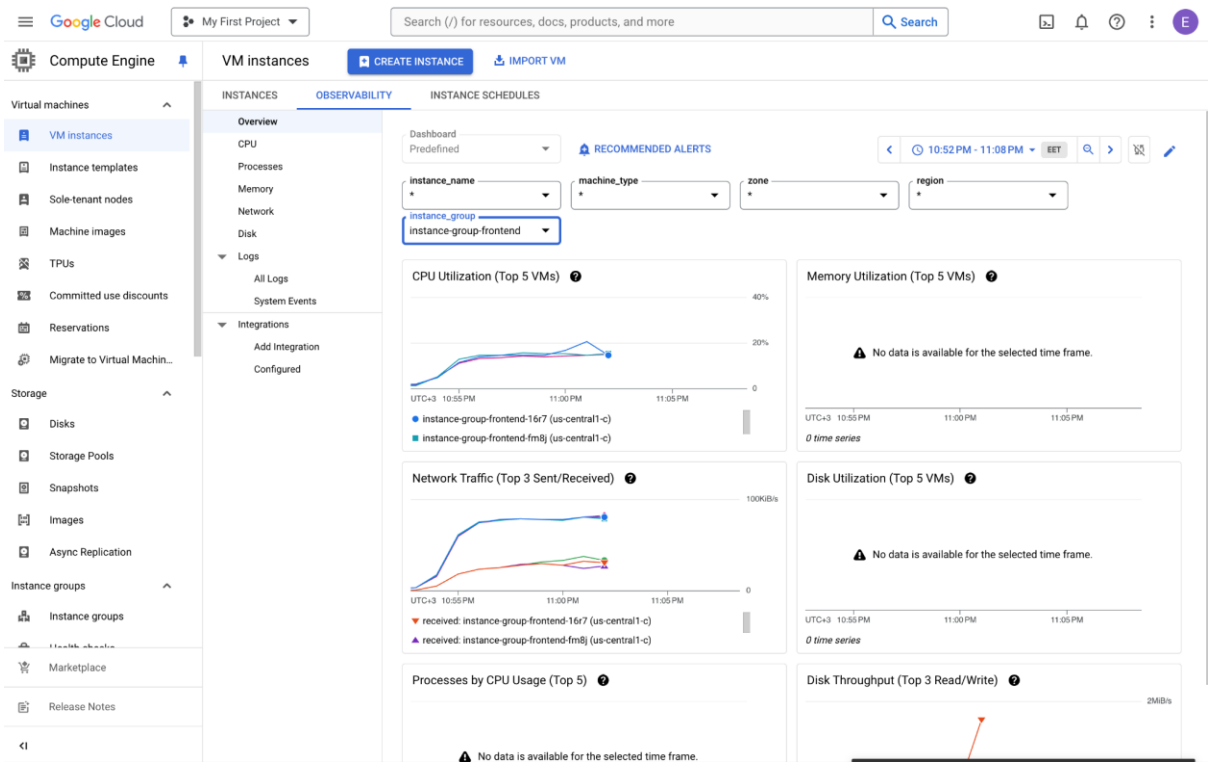


Figure 8: Frontend VMs Observability Graphs

Discussion of Experiment Results

Overview

The performance evaluation aimed to assess scalability and responsiveness of the "Ovatify on the Cloud" application under simulated high user load. Using Locust, a versatile load testing tool, and Google Cloud Monitoring, we conducted stress tests on both backend and frontend components to understand system behavior under different load scenarios.

Analysis of Backend Performance

Total Requests and Response Times (Figure 1): The graph shows a steady increase in total requests per second as the number of users ramped up by 15 users per second, which results in a corresponding increase in response times. Notably, the average response time began to rise significantly after reaching around 800 users, suggesting a potential bottleneck in resource allocation. However, scaling up of the microservices have apparently worked well and decreased the response time back to normal levels after some point, which shows that the scaling of the system was designed according to the needs.

Service-Specific Observations:

Authentication Service (Figure 2): This service exhibited peaks in request counts around 9:35 PM, which align with noticeable spikes in CPU utilization. Such patterns suggest that CPU resources become a limiting factor under high request loads, potentially leading to increased authentication delays. We also see that container instance time reaches a peak at that time interval.

Mono Service (Figure 3): The fluctuating container instance count indicates adaptive scaling in response to varying loads, we could easily see that the active number of instances change according to the number of user requests.

Recommendation Service (Figure 4): From the charts, it could be seen that along with the request count, container instance count of the service has increased, which keeps the CPU utilization at a level that is at most 60%, which prevents any potential losses.

User Service (Figure 5): Exhibits good scalability with a gradual increase in container instances to handle rising requests, peaking at the time of maximum load at the system which shows the good functioning of the service, the CPU utilization never reaches to 40% which is also a good indicator of the well system performance.

Analysis of Frontend Performance

Locust Test Graph of Frontend (Figure 7): The frontend handled increasing loads with a gradual increase in response times, maintaining under 1,000 ms until around 5,000

concurrent users. Beyond this point, response times increase sharply, indicating a potential frontend resource limits or network bottlenecks at high concurrency levels.

VM Observability (Figure 8): The observability graphs for the VMs show stable CPU utilization, suggesting that the frontend servers were not CPU-bound. We could see that CPU utilization remained at a reasonable level without any high utilization.

Cloud Function Performance

Cloud Function Observability (Figure 6): The cloud function tasked with image processing shows sharp peaks in invocations and execution times that drop quickly, indicative of effective scaling and handling sudden increases in demand. The memory utilization spike to 100% at peak times could impact performance and suggests a review of memory allocation settings.

Conclusion

The *Ovatify On Cloud* project successfully demonstrates the power of microservices architecture and cloud computing in delivering a scalable, robust music recommendation application. Leveraging Google Cloud services such as Cloud Run, Cloud SQL, Cloud Functions, Cloud Vision API, Compute Engine and Docker containerization, our system proved that it can efficiently manage thousands of concurrent users, maintaining performance integrity even under peak loads. Performance tests validated our architectural choices, highlighting effective load management and minimal latency, which together ensure an enhanced user experience.